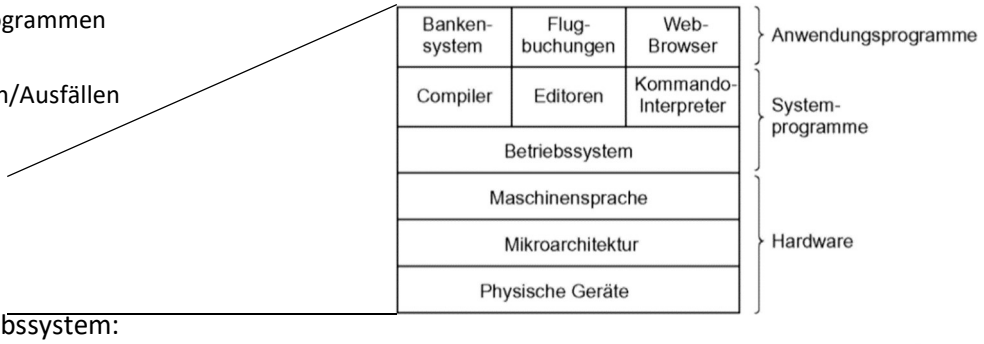
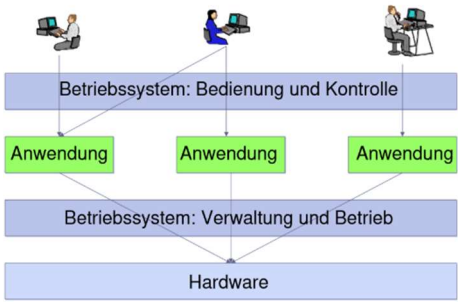
Betriebssystem:

- Aufgaben:
  - Anpassung der Maschinenwelt an Benutzerbedürfnisse
  - Regelung des Zugriffs auf Ressourcen
  - Verwaltung von Daten/Programmen
  - Effizienz (Betriebsmittel)
  - Unterstützung bei Fehldern/Ausfällen
  - Sicherheitsvorkehrungen
- Rechensystem besteht aus:
  - Hardware
  - Systemprogrammen
  - Anwendungsprogrammen



- mögliche Sichten auf Betriebssystem:
  - Virtuelle Maschine (Top-Down)
    - Bietet Programmierer abstrakte Sicht auf Hardware
    - Reale, low-level Eigenschaften werden versteckt
    - vgl. Festplatte:
      - real: Folge von Datenblöcken
      - virtuell: benannte Dateien
  - Ressourcenmanager (Bottom-Up)
    - Verwaltet Prozessoren, Speicher…
    - Koordiniert Anwendungszugriff auf Ressourcen
      - zeitliche Verwaltung: Anwendungen greifen nacheinander zu
      - räumliche Verwaltung: Anwendungen greifen auf verschiedene Bereiche zu



- Systemarchitektur:
  - wird aus einzelnen Komponenten unterschiedlichen Typs zusammengesetzt
    - Adressraum
    - Prozess
    - Signal
    - Kanal
    - …
- Historie:
  - 50er Jahre:
    - ein Programm wird von einem Prozessor abgearbeitet
    - Betriebssystemfunktion beschränkt sich auf Unterstützung I/O und Umwandlung Zahl-/Zeichendarstellung

Linux – C

- Eigene Bibliothek anlegen:
  - library-Funktion in *.c Datei schreiben (Bsp.: square.c für quadrieren)
  - Objektfile erzeugen: `gcc -c square.c` (-c um nur zu kompilieren)
  - Archivieren: `ar r libstuff.a square.o` (ar = archivieren, r = rekursiv)
  - Indizieren: `ranlib libstuff.a` (ranlib lib muss stehen, stuff.h = lib.-name)
  - zu C hinzufügen: `#include "stuff.h"`
  - Kompilieren: `gcc libtest.c -o libtest -L. -lstuff`
    - `-L.` suche im aktuellen Verzeichnis
    - `-lstuff` link stuff

make:

- manuelles Kompilieren ist oft umständlich → viele verschiedene Schalter & Eingaben
- Lösung durch `make` `make [options] [targets]`

- Skript zum automatischen kompilieren einer Datei

- Optionen:
  - `-n` build-Kommandos anzeigen ohne sie auszuführen
  - `-f <makefile>` spezielles makefile verwenden
- targets:
  - Aufbau:
    - im makefile werden targets durch rules definiert → welcher Kompilierbefehl soll ausgeführt werden

      Format:
      ```
      targets : prerequisites
          commands
          ...
      ```
    - target(s): target-Bezeichner
    - prerequisites: zum bauen des targets benötigte Files
      - beim aktualisieren der Files muss neu gebaut werden
    - commands: Rezept um das target zu bauen
  - Benennung:
    - targets gehören zu File mit dem gleine Name
    - phony-targets sind Ausnahme (erzeugen kein gleichnamiges File)
      - .PHONY als prerequisite `.PHONY: all`
      - zugehörige commands werden immer ausgeführt
- Variablen werden am Beginn des Makefiles definiert
  - vordefinierte Variablen können geändert werden →

Name	Description
CXX	C++ compiler command
CXXFLAGS	C++ compiler options
LDFLAGS	linker options

- plattformübergreifend: automatisches Erstellen des makefiles
  - automake (Makefile.am → Makefile.in), Makefile.am hat nur plattformübergreifende Optionen
  - configure (Makefile.in → makefile) aus `configure.ac` + autoconf

Debugging:

- gdb: Schrittweises Durchlaufen eines Codes (siehe gnu.org/s/gdb & VL07 / 15)
- Logs: beinhalten Speicherbild des abgestürzten Prozesses, crash-Tool kann Speicherbild des abgestürzten Kernels auslesen

Dateiabhängigkeiten:

- eine Datei hängt von einer anderen ab, wenn die andere Datei zum erstellen der 1. Datei benötigt wird

Übergabe von beliebig vielen Variablen:

- ...-Parameter übergibt beliebig viele Variablen:

- Interne Verarbeitung mit `va_start`, `va_arg` und `va_list` (aus `stdarg.h`)

Beispiel: beliebig viele Zahlen addieren

```
#include <stdarg.h>
int add(int zahlen, ...) {
   va_list zeiger;   int zahl;
   va_start(zeiger, zahlen);
   do {  zahl = va_arg(zeiger,int);
         zahlen += zahl;  } while(zahl != 0);
   va_end(zeiger);  return zahlen;  }
```

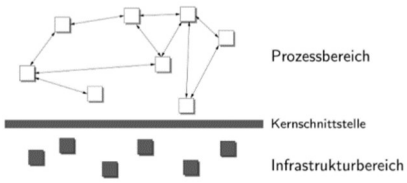
- Aufruf z. B. mittels

```
printf("%d\n",add(11,12,13,0));
```

- **Regel**: *mind.* ein Parameter muss bekannt sein

Kernel

- Prozessor ist selbst ein Betriebsmittel → Zugriffszeit von Programmen wird vom Betriebssystem eingeteilt
- Zusammenarbeit: Prozessor – Betriebsmittel – Betriebssystem – Programme
- OS-spezifischer Teil in Systemsoftware ist der OS-Kernel
- Betriebssystem ist zuständig für:
  - CPU- / Gerätecontroller:Hardware-Bus-Verbindung ermöglicht Zugriff auf gemeinsamen Speicher
  - Betrieb von CPU und Geräten: Speicher- und Rechenzeit verwalten
- Unterscheidung in zwei Bereiche:
  - Prozessbereich: eigentliche Funktion des Betriebssystems
  - Kern(bereich): Infrastruktur für Prozesse



Benutzer- / Systemmodus:

- Benutzermodus: zur Ausführung normaler Programme
- Systemmodus (Privilegiert): Lesen/Schreiben von Registern, Speicherverwaltung
- Unterscheidung zum Schutz von BS und internen Daten
  - Bit im Programmstatuswort gibt aktuellen Modus an
  - durch BS-Dienst kann PSW-Bit geändert werden
- verschiedene Betriebssysteme führen verschiedene Befehle im Benutzermodus aus

➔ Alles was im Systemmodus (Kernmodus) läuft gehört zum Betriebssystem

Interrupts:

Arbeit des CPUs wird unterbrochen, um andere Routine im Systemmodus auszuführen

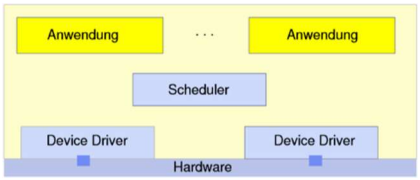
- asynchron: IRQ (Request) von Peripherie oder PageFault (Zugriff auf Speicher außerhalb des RAM)
  - Direct Memory Access: RAM-Zugriff unabhängig von der CPU
    - Netzwerk- / Soundkarten
    - blockweiser Transfer von Daten (statt byteweise)
- synchron: ausgelöst durch Trap/Exception zur Verarbeitung von Fehlern (SIGSEGV) / Signalen (SIGINT)
  - siehe signal.v (VL 05 / 3)
  - Trap / Exception
    - Prozess wird unterbrochen & Signal an Trap-Handler übergeben
    - Verarbeitung technischer Fehler

Mikro- / Makrokernarchitektur:

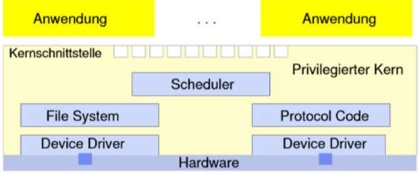
- Mikrokern-Architektur: nur essenzielle Betriebssystem Funktionen im Kern enthalten
- Makrokern-Architektur: z.B. UNIX / Windows, auch Dateisystem etc. im Kern realisiert

Betriebssystem-Klassen:

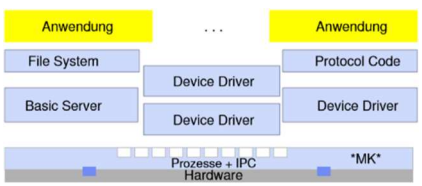
- Monolithische Systeme:
  - keine strenge Trennung von Programmen und BS → rufen sich gegenseitig auf
  - nur für kleine, statische Betriebssystem geeignet, z.B. MS-DOS



- Makrokern (Monolithischer BS-Kern):
  - Trennung Anwendungen und BS
  - geschichtetes BS → hierarchische Anordnung von Funktionen
  - problematische Schichtänderungen: große Auswirkung, schwer zu verfolgen



- Mikrokern:
  - Kern umfasst nur Prozessmanagement (Scheduling, Dispatching, …)
  - externe Teilsysteme: Treiber, Dateisystem, …



- Mikrokernarchitektur:
  - Vorteile:
    - Klare Kernschnittstelle
    - da Dienste außerhalb des Kerns → Sicherheit, Stabilität, Flexibilität, Erweiterbarkeit, Portierbarkeit
  - Nachteile:
    - schlechtere Performance wegen aufwändigerem Zusammenspiel der Komponenten

System-Calls:

- erfordern den Wechsel in Kernel-Modus, Spezifikation einer OS-Routine, Parameterübergabe auf Stack

Remote Procedure Call (RPC)

1. Funktion wird aufgerufen: wirkt lokal, liegt aber auf Server
2. **Marshalling**: stub (auf Client) vereint Parameter, verallgemeinert sie, sendet Nachricht an Server
   - Verallgemeinerung von Param.: flache, pointerfreie Darstellung (pointer → lokale Adressen)
3. Server empfängt Nachricht, konvertiert Parameter (unmarshalling) in auf Server-Seite lesbare Darstellung
4. Aufruf der Server-Funktion, zurückschicken des Ergebnisses (nach marshalling)
5. Client empfängt das Ergebnis → verarbeitet es nach dem **Unmarshalling** weiter

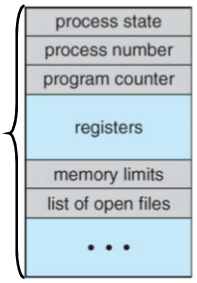
SunRPC:

- SunRPC wird von keiner Programmiersprache unterstützt und braucht deshalb einen Pre-Compiler (**rpcgen**)
  - Input: remote procedures in IDL (interface definition language)
  - Output: server main routine, client stub functions, header file, data conversion functions
- Interface erstellen:

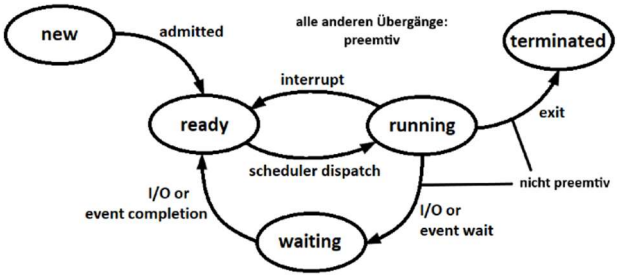
```
program DATE_PROG {              //program = Interfacedefinition
     version DATE_VERS {
          long BIN_DATE(void) = 1;                 //Prototyp #1
          string STR_DATE(long) = 2;               //Prototyp #2
     } = 1;                                       //Versionsnummer
} = 0x31423456;                                   //Programmnummer
```

Prozesse

- Betriebsmittel:
  - privaten Stack, Heap, Code & Daten
  - Registerinhalte werden beim Kontextwechsel (schalten zwischen Prozessen) zwischengespeichert
  - Betriebssystem sicher Prozessen Konsistenz zu (bzgl. Zuständen & Betriebsmitteln)
    → Context Switches haben hohen Aufwand
- Prozesskontrollblock = PCB
  - Programmzähler, Registerinhalte, Scheduling-Infos, Memory-Management, Account-Daten und Prozesszustand



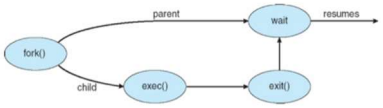
- Prozesszustände
  - new, ready, running, waiting, terminated
  - Zustand terminated ist entgültig
  - Context switches haben wegen ihrem hohen Aufwand eine lange „idle“ -time, in der keine Prozesse beschäftigt werden
  - preemtives Scheduling: Scheduler greift ein
  - nicht-preemtiver S.: Programm entscheidet ob Prozess geändert wird



- Adressräume
  - beliebig vilee logische Adressräume können gebildet werden
  - ein Prozess hat einen Adressraum ODER
  - mehrere Prozesse teilen sich einen Adressraum → shared memory (Heap)
    - einfachste Möglichkeit der Kommunikation zwischen Prozessen
    - ein Speicher-Segment wird dem logischen Adressraum mehrerer Prozesse zugeordnet
    - Zuweißung: `shmget();` Anbindung: `shmat();` Trennung: `shmdt();`
- Prozesstypen
  - I/O-Bound: meiste Zeit wird für E/A-Operationen aufgewendet, nur sehr kurze CPU bursts
    - z. B. grafische Oberfläche, Bildverarbeitung
  - CPU-Bound: Prozesse, die wenige, sehr lange CPU-bursts haben
    - z.B. Benchmarks, Numerik (Matlab)
  - Verarbeitung bzw. Implementierung einer Problemlösung entscheidet den Prozesstyp
    - bei schlechter Verarbeitung kann CPU- zu I/O-Bound werden

Prozesserzeugung:

- Erzeugung:
  - `fork ( )` erzugt Kindprozess → Kopie aus dem Elternprozess, gleiche Variableninhalte, Änderungen nur lokal
    - `fork ( )` gibt Process-ID zurück, kein Kind: `fork ( ) = 0`
  - `wait ( )` → Elternprozess wartet auf Terminierung Kindprozess
  - `exit ( )` → Kindprozess wird terminiert



- halb-duplex Pipes:
  - einseitiger Kanal, Richtung, die nicht verwendet wird, wird geschlossen

```
#include <stdio.h>
#include <unistd.h>
int main(int argc, char **argv)  {
 int fromChild[2], toChild[2];  char buffer[50];
 pipe(toChild);  pipe(fromChild);
  if (fork( ) == 0)  {/* child: */
   close(fromChild[0]);  close(toChild[1]);
   write(fromChild[1], "ich bin das Kind", 17);
   read(toChild[0], buffer, 18); printf("child received: %s\n", buffer);
  } else  {  /* parent: */
   close(toChild[0]);  close(fromChild[1]);
   write(toChild[1], "ich bin Dein Vater", 18);
   read( fromChild[0], buffer, 17);  printf("parent received: %s\n",
         buffer );
  }  return 0;
}
```

2 Richtungen (0 = Lesen, 1 = Schreiben)

Kanäle können beschrieben & ausgelesen werden

Nicht benötigte Kanäle schließen

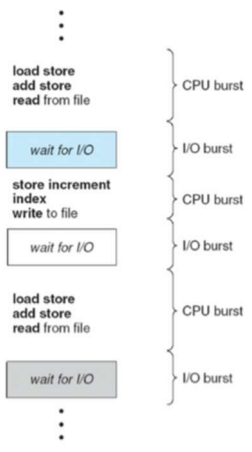
- fork-Bombe:
  - Endlos-Schleife mit `fork( )` → unendlich viele Kindprozesse, alle Betriebsmittel werden belegt

Prozesse und virtuelle Prozessoren:

- Prozesse konkurrieren um Prozessor, da mehr Prozesse als Prozessoren vorhanden sind
  - ➔ Process contention
- Context Switch:
  - realer Prozesor: aktives Betriebsmittel zur Programmausführung
  - virtueller Prozessor: Folge von Nutzungsabschnitten realer Prozessoren zur Programmausführung
- Prozessor arbeitet kontinuierlich an Prozessen, aber Prozesse „warten, starten, halten an und werden beendet“
- dynamische Auswahl, welcher Prozess als nächstes bearbeitet wird
  - mögliche Kriterien: Nummer, Ankunftsreihenfolge, Priorität

Scheduling Festlegung der Reihenfolge von Prozessen

- Queues:
  - Job Queue: Prozesse des Betriebssystems
  - Ready Queue: Prozesse im Hauptspeicher, für Ausführung bereit (unterbrochen)
  - I/O Device Queues: Prozesse die auf Eingabe / Ausgabe warten → verschieden Queues für Festplatten, Eingaben etc.
- Arten:
  - Long-Term Scheduler: welche Prozess kommt als nächstes in ready-Queue
    - bestimt Anzahl der gleichzeitig aktiven Anwendungen (Multiprogramming)
    - im Sekundentakt aktiv
  - Short-Term Scheduler: welcher Prozess von Ready-Queue kommt als nächstes in CPU (Responsivität des Systems)
    - muss schnell sein → im Millisekundentak aufgerufen
  - Medium-Term Scheduler: Grad des Multiprogramming bei hoher Last reduzieren → hilft Long-Term Scheduler
  - beobachten des Scheduling mit pidstat
- Strategien:
  - Scheduling-Prioritäten mit nice-Level
    - nice-Level zwischen -20 und 19 → wird Prozess zu Beginn zugeteilt
    - Vergleich der Prioritäten: $\frac{20-p1}{20-p2}$ → Ergebnis gibt an wieviel Rechenzeit p1 im Gegensatz zu p2 bekommt
      - Bsp.: Priorität p1 = 15, Priorität p2 = 0 → $\frac{20-15}{20-0} = \frac{1}{4}$ → p1 bekommt ¼ Rechenzeit von p2
    - Problem Priorisierung:
      - Starvation mancher Prozesse (Priorität zu niedrig = keine Rechenzeit)
      - Lösung: Je älter Prozess wird, desto höher seine Priorität
  - Shortest-Job First Scheduling (SJF):
    - „Job“ mit kürzestem burst (I/O-Bound oder CPU-Bound) bekommt höchste Priorität
  - Round Robin:
    - jeder Prozess erhält gleichen Anteil (10-100ms Rechenzeit / time quantum)
    - danach Prozess preemtiv in ready-Queue
      - n = Anzahl Prozesse, q = Quantum; Prozess wartet max. (n-1)*q Zeiteinheiten
    - schlechtere turnaround time, bessere response time
- Kriterien für Auswahl Verfahren
  1. CPU utilization → maximale Auslastung
  2. throughput → Anzahl der abgeschlossenen Prozesse
  3. turnaround time → Zeit bis zur Ausführung eines Prozesses
  4. waiting time → Wartezeit Prozess in der ready-queue
     - □ (Summe Startzeiten)/Anzahl Prozesse = Durchschnitt
     - □ optimal bei SJF -Scheduling
  5. response time → Antwortzeit nach erster Anfrage
  - Convoy-Effekt bei FCFS (first come first serve): kurze Prozesse warten auf lange
  - Schätzung der Länge des CPU-Bursts durch exponentielle Glättung
    1. t_n Länge des n-ten CPU-Bursts
    2. T_{n+1} vorhergesagte Länge des nächsten CPU-Bursts (T_n = n-te Vorhersage)
    3. α $0 \leq \alpha \leq 1$
    4. $T_{n+1} = \alpha * t_n + (1 - \alpha) * T_n$



POSIX Threads

leichtgewichtige Prozesse, also unabhängige Ausführungsabfolgen einer Software → ein UNIX-Prozess kann mehrere POSIX-Threads ausführen

Code:

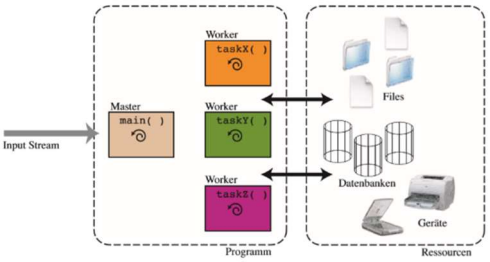
`#include <pthread.h>` `gcc -lpthread` (evtl. `-D_POSIX_PTHREAD_SEMANTICS`)

`pthread_create (t, attr, code, args)` Erzeugung & Starten eines Threads, z.B. (&t1, NULL, runner, &n1)

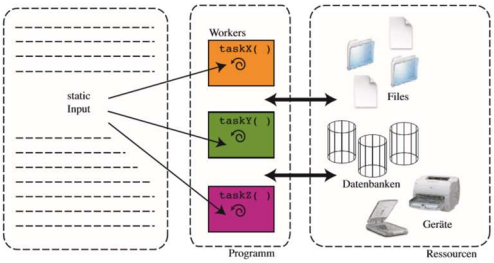
`pthread_join(t, res)` blockiert ausführenden Thread bis Terminung eines anderen Threads t

Programmiermodelle:

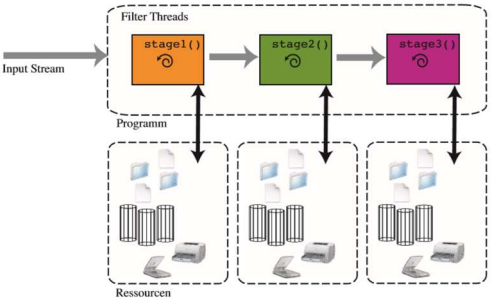
- Compute Farm:
  - Master: wartet auf request, dynamisch neue Threads
  - Worker: verarbeitet Task, Abarbeitung spezifizierter Code
  - Prinzip: Abhängigkeiten innerhalb Worker minimieren



- Workcrew / Peer-to-Peer
  - Master: generiert Workerthreads, wartet auf Terminierung
  - Worker: verwenden geminesame, statische Datenquelle
  - für Verfahren mit unveränderlicher Eingabe (Datenbanken)



- Pipeline
  - für Fertigungsprozesse
  - große Menge unabhängiger Eingabedaten durch Abfolge von stages verarbeitet
  - Schachtelung einer Farm / Workcrew innerhalb von stages
    → (De-)Multiplexing
  - maximaler Durchsatz auf Dauer der aufwändigsten stage begrenzt



Mutex:

- wechselseitiger Zugriff auf Ressourcen → Mutual Exlcusion (MutEx) immer: `pthread_mutex_...`
  1. Anlegen & Initialisieren
     - `..._t *var` und `..._init(var, NULL)`
  2. Vor Eintritt in kritischen Bereich Sperre anfordern
     - `..._lock(var)` und `..._trylock(var)`
  3. Nach Abarbeitung wird Sperre freigegeben
     - `..._unlock`
- Threadkoordination mit Condition Variablen immer: `pthread_cond_...`
  - `..._t` und `..._init` zum initalisieren
  - `..._wait` zum warten auf Sperre von anderem Thread, gibt Sperre frei
  - `..._signal`/`..._broadcast` Aufruf zur Benachrichtung über Statusänderungen, gibt Sperre frei
    → wait- & signal-Blöcke werden immer von lock / unlock umschlossen
  - Threads warten nach dem FIFO-Prinzip auf Freigabe von Sperre, Prioritäten können vergeben werden
  - wenn wait / signal / broadcast Sperre nicht freigeben würde → Deadlock

Problematiken bei Threads:

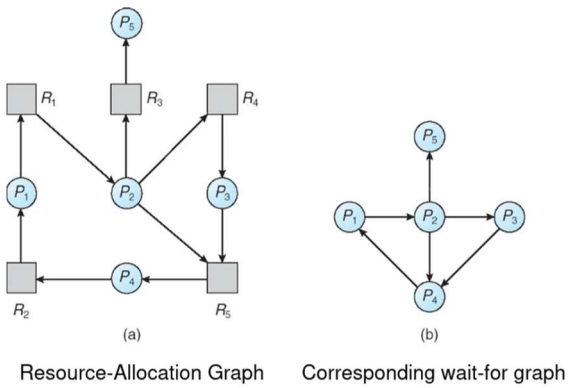
- Deadlock
  - mehrere blockierte Prozesse, die Ressorucen halten und darauf warten, von anderen Prozessen (im gleichen „Set“) Ressourcen zu erhalten
  - Bedingungen: (müssen für Deadlock alle erfüllt sein)
    1. serielle Benutzung von Ressourcen (mutual exclusion, nur jeweils eine Ressource für einen Prozess nutzbar)
    2. Inkrementelles Ressourcen-Erlangen (beim warten auf Ressource wird gehaltene nicht freigegeben)
    3. Keine Preemption (Prozess kann gehaltene Ressource nicht genommen werden)
    4. Wartezyklus (Prozesskette, in der jeder Prozess die Ressource hält, auf die sein Vorgänger wartet)
  - Lösungen:
    - Detection & Recovery → Terminierung einzelner Prozesse & aufheben 3. Bedingung
    - asymmetrisches Verhalten → aufheben Bedingung 4, kann zu Starvation führen und ist unfair
- Starvation
  - ein Prozess kann auf Kosten seines Nachbarn immer wieder die Ressource erlangen
  - Bedingungen:
    - LIFO-Scheduling → untere Prozesse kommen bei hoher Last nie dran
    - ungünstige Priorisierung → B höher priorisiert als C, aber C hält Ressource von A (Priority Inversion)
  - Lösung:
    - Prioritäts-Vererbung: A gibt Priorität an C ab
    - Timeout: vor Ressourcenaufnahme Prüfung der Wartezeit des Nachbartasks
    - externer Agent: nicht alle Prozesse gleichzeitig starten bzw. laufen lassen
- Busy Waiting
  - Prozesse, die andauernd eine Bedingung prüfen

Ressource Allocation Graph:

- Graph (V, E)
  - V ist Menge aus P (Prozesse) und R (Ressourcentypen R_i, aufgeteilt in Instanzen R_j)
  - E:
    - Anforderungskante: $P \rightarrow R_i$ Prozess fordert Ressource an
    - Zuweisungskante: $R_j \rightarrow P$ Ressourceninstanz is Prozess zugeteilt
    - Claim Kante: $P \rightarrow Ri$ gestrichelt, Prozess könnte Ressource anfordern
- Fakten:
  - kein Zyklus → kein Deadlock
  - Zyklus:
    - eine Instanz / Ressource = Deadlock
    - mehrere Instanzen / Ressource = möglicher Deadlock
  - Safe State: System in sicherem Zustand → kein Deadlock
  - Unsafe State: Systein in unsicherem Zustand → Deadlock möglich
    - Avoidance: Verhindern von Deadlocks
      - eine Instanz / Ressource → Resource Allocation Graph
      - mehrere Instanzen / Ressource → Bankieralgorithmus

Safety Algorithmus:
1. suche $Need_i \leq Available$
2. $Available = Allocation_i + Available$
3. Schritt 1 & 2 wiederholen bis alle abgearbeitet sind

- Behandlung von Deadlocks:
  - Sicherstellung, dass kein Deadlock-Status erreicht wird
  - Erreichen eines Deadlock-Statusses erlauben, überprüfen ob Deadlock vorliegt → Recovery
  - Ignorieren → Annahme, dass keine Deadlocks auftreten
- wait-for Graph:
  - Knoten sind Prozesse
    - $P_i \rightarrow P_j$ P_i wartet auf P_j
  - n^2 Operationen mit n = Anzahl der Ecken
- Deadlock Recovery:
  - Process Termination:
    - alle durch Deadlock blockierte Prozesse abbrechen
    - nach jedem Prozessabbruch prüfen ob Deadlock aufgelöst
  - Resource Preemption:
    - Select a victim → Rollback → Starvation



Resource-Allocation Graph Corresponding wait-for graph

Dienste und Betriebsmittel

Dämonen:

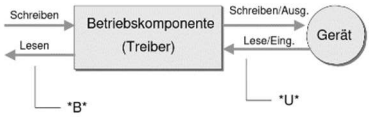
- unterstützender Dienst, durch den das Betriebssystem Anwendungen Ressourcen zur Verfügung stellt
  - läuft dauerhaft nebenbei
  - z.B. `initd` → PID 1, darf nicht beendet werden
  - manche Dämonen sind unsterblich → können nur mit SIGKILL beendet werden
    - z.B. fork-Bombe `while(1) { fork ( ); }`

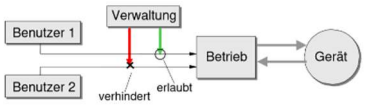
Zombies:

- können wie Dämonen nicht getötet werden
- im Gegensatz zu Dämonen sind Zombies völlig nutzlos, bereits „tot“ aber steht noch in der Prozesstabelle
- entsteht durch Terminierung eines Kindprozesses, der aber KEIN `wait( )` im Elternprozess zur Folge hat
  - Kindprozess verbleibt als Zombie
  - entfernen der Zombies durch Terminierung des Elternprozesses
- benötigt kaum Ressourcen, wirkt sich negativ auf Scheduling aus

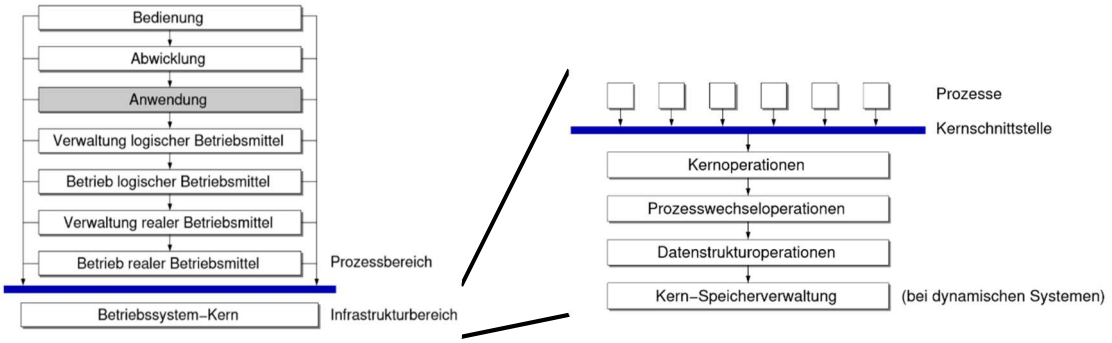
Betriebsmittelnutzung:

- Arten von Betriebsmitteln
  - Logische Betriebsmittel:
    - aus organisatorischen Gründen ausgedacht, Realisierung durch reale BM
    - Bsp.: Datei, Fenster
  - Physikalische Betriebsmittel:
    - Real vorhanden
    - Bsp.: Platte, Bildschirm





- Umgang mit Betriebsmitteln
  - Betrieb: Tatsächliche Nutzung, z.B. Datentransport
  - Verwaltung: Zugriff erlauben (Contention) → beansprucht BS mehr als Nutzung



- Kernschnittstelle:
  - Systemcalls: Betriebssystemfunktion, die von Benutzerprogrammen aus aufgerufen werden können
  - Beispiele in Unix:
    - Prozessmanagement: fork ,exit
    - Dateimanagement: open, read/write
    - Verzeichnismanagement: mkdir , unlink (Dateiname entfernen)
    - Verschiedenes: chmod (Zugriffsrechte ändern), kill (Signal an Prozess)

Speichermanagement

Memory-Management-Unit:

- Hardware, die virtuelle in phyissche Speicheradressen umrechnet, User arbeiten aber mit logischen und nicht physischen Speicheradressen
- Dynamic relocating:
  - Dynamic Loading Routine wird erst geladen wenn sie aufgerufen wird
  - Dynamic Linking Routine wird bei Ausführung an bestimmte Stelle im Speicher geschoben
  - Swapping Temporärer Austausch eines Prozesses in „Wartespeicher“ (Ready Queue)
- Dynamic Storage-Allocation:
  - First-fit: Speichern an erster Stelle wo Platz ist → am schnellsten
  - Best-fit: kleinste Stelle die groß genug ist → kleinste „Lücke“
  - Worst-fit: größte Stelle wird belegt → größte „Lücke“

Framentierung:

- External: kein zusammenhängender Speicherbereich für einen request, aber genug freier Platz
  -
- Internal: zugeteilter Speicherplatz ist mehr als angefragt
  - Paging Addresse geteilt in page number und offset
    - Physischer Speicher wird in **frames** fester Größe unterteilt
    - Logischer Speicher wird in **pages** gleicher Größe unterteilt
    - für ein Programm mit Größe n pages braucht man n freie Frames
    - Arten:
      - Hierarchical logische Addresse wird in verschiedenen page tables aufgeteilt
      - Hashed virtuelle page-Nummer wird in eine page table gehashed
      - Inverted ein Eintrag pro page
    - Demand Paging: Page wird nur in Speicher geladen wenn sie gebraucht wird
      - page fault: Referenz auf Seite, die nicht im Speicher ist
        - p = Page Fault Rate, p(time) = page fault service time
        - Effective Access Time = (1-p) * memory Access + p(time)
      - Belady’s Anomaly: more frames = more page faults

File System:

- Structure:
  - none: sequnce of words, bytes
  - Simple record structure: Lines, fixed length, variable length
  - complex structure: formatted document, relocatable laod file